AFRL-RB-WP-TR-2008-3061

AVEC: A COMPUTATIONAL DESIGN ENVIRONMENT FOR CONCEPTUAL INNOVATIONS

Maxwell Blair

Design and Analysis Methods Branch
Structures Division

FEBRUARY 2008
Final Report

AIR FORCE RESEARCH LABORATORY
AIR VEHICLES DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7542
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

AFRL-RB-WP-TR-2008-3061 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

*//Signature//

MAXWELL BLAIR
Aerospace Engineer
Design and Analysis Methods Branch
Structures Division

//Signature//

GREGORY H. PARKER
Deputy Chief
Design and Analysis Methods Branch
Structures Division

//Signature//

DAVID M. PRATT, Ph.D.
Technical Advisor
Structures Division

*Disseminated copies will show "//Signature//" stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
February 2008	Final	01 October 2003 – 30 September 2006

4. TITLE AND SUBTITLE
AVEC: A COMPUTATIONAL DESIGN ENVIRONMENT FOR CONCEPTUAL INNOVATIONS

5a. CONTRACT NUMBER
In-house

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
0602201

6. AUTHOR(S)
Maxwell Blair

5d. PROJECT NUMBER
A03H

5e. TASK NUMBER

5f. WORK UNIT NUMBER
0B

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Design and Analysis Methods Branch (AFRL/RBSD)
Structures Division
Air Force Research Laboratory, Air Vehicles Directorate
Wright-Patterson Air Force Base, OH 45433-7542
Air Force Materiel Command, United States Air Force

8. PERFORMING ORGANIZATION REPORT NUMBER
AFRL-RB-WP-TR-2008-3061

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory
Air Vehicles Directorate
Wright-Patterson Air Force Base, OH 45433-7542
Air Force Materiel Command
United States Air Force

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/RBSD

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-RB-WP-TR-2008-3061

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES
PAO Case Number: AFRL/WS 06-0535, 23 Feb 2006. Report contains color.

14. ABSTRACT

This report summarizes programming techniques that aid multidisciplinary design programmers in developing computational designs that measure AFRL technology effectiveness. These techniques have been collected into an object-oriented design environment. The Air Vehicle Environment in C++ (AVEC) prototypes a practical approach toward computational design. Design innovators will benefit from AVEC at one of three levels. These three levels target (a) the end user through interactive operations and file I/O, (b) the object-oriented programmer through a compiled library of properly documented and inheritable objects, and (c) the AVEC developer who wishes to enhance AVEC capability with modifications to the source code. The pilot code presented here focuses on parent-child relationships, automated dependency management, geometry, meshing and analysis. All together, the overall capability leads to design variant management that will populate a response surface model and thereby address design optimization. The target SensorCraft design mission involves a suite of aeroelastic concepts with geometric non-linearity, in the form of non-linear coupling, large deformations and follower forces.

15. SUBJECT TERMS
computational aeroelasticity, aerothermoelasticity

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Maxwell Blair
Unclassified	Unclassified	Unclassified	SAR	46	19b. TELEPHONE NUMBER *(Include Area Code)* N/A

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

Table of Contents

LIST OF FIGURES

LIST OF TABLES

FOREWORD

The work reported here supports the in-house research mission of the Air Force Research Laboratory, Air Vehicles Directorate, MultiDisciplinary Technology (MDT) Center. AVEC is a prototype software programming environment based on ANSI standard C++. AVEC arises from research to support future Computational Design efforts in support of Air Vehicles Directorate integrating concepts and the AFRL technology mission.

Parallel programming environment developments are cited in this report. Specifically, this report highlights the in-house research that followed the cost-shared Dual-Use development effort, "Scenario-Based Affordability Assessment Tool" (SBAAT), contract F33615-00-2-3055.

The target application for the AVEC environment is any computational design model that requires both the computational efficiency of a compiled code and geometric design versatility associated with object-oriented programming. Extensions to the AVEC environment can be distributed in the form of shared libraries of class structures. Such extensions should address various geometric and meshing needs, analysis needs, graphical renderings, both static and dynamic. AVEC supports the integration of these class structures with automated dependency management to form an interactively managed design process.

This report was cleared for Public Release by AFRL Public Affairs. Disposition Date: 23 February 2006. Document Number AFRL/WS 06-0535.

The author extends his appreciation to Dr. Richard Snyder for his knowledge and support in compiling AVEC in standard ANSI C++ Object-Oriented Software Language

1. Introduction

Aerospace design is tightly integrated with computational models that represent a variety of physical phenomena. Innovative designs may be realized with innovative software programming environments. Successful software developments follow rigorous software requirements. This paper presents a number of tested software concepts in the form of a pilot code that is one step from forming rigorous software requirements.

A good Computational Design environment integrates the fidelity of computational physics with the speed of conceptual design. Reference [1], provided an in-depth overview of the motivation for computational design development. This was followed by a description of the pilot code and the software innovations that were tested. AVEC is a prototype for an open-source "*Air Vehicle Environment in C++*".

The author's goal in this paper is to put forth engineering software needs to a software development community. These needs have been developed after a number of years of experience. This paper describes a number of desired features for a library of inheritable Computational Design classes. For instance, one programming capability that arises from class inheritance is the integration of geometric and non-geometric parameters in a single class entity. Thus, a engineer programmer can develop integrated class entities with component geometry that directly supports component analysis. For example, the AFRL is interested in development of conformal load-bearing antenna structures. The performance of the antenna is tightly dependent on the electrical currents that run though conductors that conform to the shape of various aircraft parts. Thus, the design of antennas benefit with software classes that integrate geometric parameters with electrical parameters.

A Design Environment is a good thing if it manages an otherwise intractable system design problem. A Computational Design Environment is appreciated to the extent

- **Data is timely:** Mundane operations are automated, thus freeing the designer to focus on intuitive aspects.
- **Data is relevant:** System performance is based on reliable integrated models that address all physics with sufficient fidelity and an appropriate level of uncertainty to account for a lack of fidelity.
- **Design space is scalable:** The ability to manage a very large family of designs significantly reduces uncertainty. The designer is constantly searching for the broadest and most comprehensive design space possible.
- **Designs are optimized:** Going the next step beyond design insight to include an automated search of design space.
- **Discovery is achievable:** The choice and form of design variants is not restricted to historical precedent.

The AVEC prototype is preceded by a number of noteworthy developments. ICAD is described in reference [2] as a pioneer example of an object-oriented design environment based on a generative model that transforms input specifications into a product design through a number of relevant procedures. Two early efforts leading to Computational Design are described in References [3] and [4]. Reference [3] prototypes a design hierarchy data model in Fortran that is roughly replicated in this work in the form of an adaptable dependency management class.

Reference [4] prototypes dependency management in a environment for the optimization of an object-oriented conceptual design model. In more recent years, we see Reference [4] leading to a capability for the optimization of complex systems described in Reference [5]. Reference [3] has lead to the Computational Design Capability described in Reference [6]. Reference [7] represents a significant Computational Design capability based on scripted freeware (free software) with application to the design of a hypersonic concept. DAKOTA is described in reference [8] as a general purpose freeware toolkit written in C++. It serves as an interface between codes while addressing optimization and uncertainty quantification. The design model of Reference [9] for the optimization of a High-Altitude Long-Endurance (HALE) concept with joined-wings was based on a commercially licensed scripted environment. This aeroelastically trimmed design model was complex with geometric non-linear structures and follower forces. Certainly, a number of other Computational Design Environments are also in development as they guide developmental decisions leading to future flight concepts. These environments serve the designer as an integration tool but with significant differences in their program structure and data process mechanisms.

AVEC is in an early stage, and will require another year or more of development before practical design applications are published. This paper will serve as a reference in order to document the underlying principles of the AVEC environment. This paper describes a number of software techniques already established in AVEC that will help aerospace research specialists create, assemble and manage classes for a collection of computational models with design applications. Going beyond the end user perspective, a C++ programmer today can readily inherit and modify AVEC classes related to dependency management and geometric rendering. End users will appreciate the parametric nature of design models that become established in AVEC.

No one programmer can be expected to anticipate all software-driven design innovations. Practical sharing strategies are critically important to achieve and maintain technical leadership. Clearly, sharing happens with commercially-developed software and is driven by financial considerations. When creativity is the dominant concern, open-source is a more effective sharing strategy. Open-source development draws on a potential base of many tens of thousands of innovative computer programmers (in the U.S.) and many more engineering specialists with computational skills. Open source software begins with a tight nucleus of highly trained and inspired developers who can deliver a reliable prototype code that can be appreciated by a relatively large base of computational designers. Software innovators can be freely guided by design engineers who develop innovative designs.

This author is highly motivated to pursue open-source distribution. Open-source succeeds with a good software library that is free, reliable, scalable, useful and unique. Open-source distribution will be possible to the extent the classes are documented and validated on test cases. An envisioned Computational Design application is described in the following section.

2. SensorCraft Background

AFRL maintains a number of airborne concepts that serve as integration concepts for any number of technologies. These airborne concepts provide a context for both the technology developer and the technology investor. These concepts address a variety of missions that include Long Range Strike, Space Access, Mobility and others.

One such concept is the AFRL SensorCraft as described in Reference [10]. It is a conceptual flying antenna farm whose design intent is to replace several flight systems (currently in service) with a single integrated system. The technologies that come out of the AFRL SensorCraft program will benefit both new and existing systems.

The AFRL SensorCraft technology-development program delivers next-generation ISR (intelligence, surveillance, reconnaissance) technologies in the context of system level integration of a HALE (High Altitude Long Endurance) concept.

Just about everything that goes into a SensorCraft concept is a new technology. For instance, AFRL is looking at advanced structural concepts, multifunctional antenna structures, active aeroelastic wing technology, active boundary layer control etc. All these new technology developments are influenced by the choice of integrating concept. The effectiveness of the overall system is influenced by the choice of technology suite. The cartoons depicted in Figure 1 give an idea of a few of the many configuration variants that come into consideration.

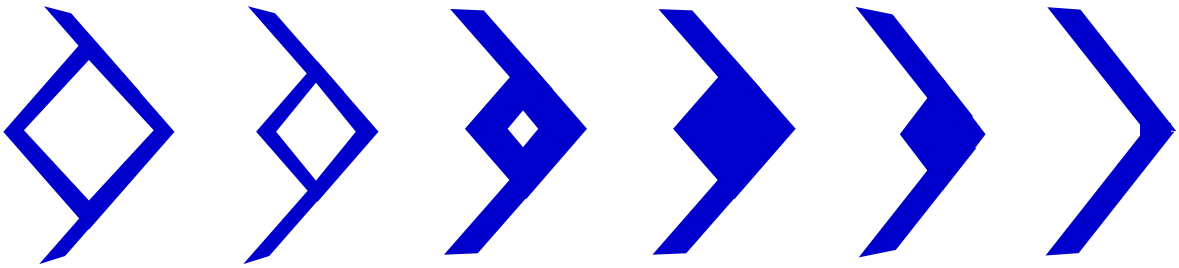


Figure 1: Sample of SensorCraft Design Variants

Going beyond a purely academic design exercise, the AFRL is highly motivated to reduce developmental costs through Certification-by-Analysis (CBA). CBA uses software tools as a way to reduce uncertainty and risk, thereby supporting the transition from concept to testing.

The unconventional joined-wing concept is an example in which global geometric non-linearity dominates the critical structural failure modes. Consequently, a structural weight penalty is associated with non-linear mechanics. Rather than dismissing the joined-wing, we have elected to view non-linearity as an opportunity to create a weight-competitive design. A thorough understanding of practical joined-wing design requires a Computational Design Environment to drive a daunting certifiable design study without a major investment.

3. SensorCraft Design Challenges

HALE concepts are complex. A high-level computational design environment such as the envisioned AVEC system is required to manage, analyze and optimize the myriad of disciplines, technologies and design variants that play in the overall system optimization of any SensorCraft concept.

3.1 Disciplines:

Three traditional disciplines of Aerodynamics, Structures and Controls interact mutually in the overall performance of a HALE concept. Traditionally, structural designers strive to minimize the weight of the vehicle. Aerodynamic designers strive to minimize the drag of the vehicle. Controls address equilibrium (trim), controllability and stability. Since all three disciplines interact mutually, all three disciplines should be managed in one environment. Aerodynamics and controls influence structural loads. Structural weight affects aerodynamic drag (induced) and trim. Control effectors affect structural weight and aerodynamics. Reference [9 Blair] and [11 Craft] and [12 Smallwood] addresses an envisioned environment that addresses minimum weight, minimum drag, sensor performance for an aeroelastically trimmed HALE concept.

Any HALE concept is inherently large, light and flexible. Two non-linear contributors dominate HALE design, structural geometric non-linearity and follower forces. Any span-braced (e.g. joined-wing, strut-braced wing, tail-braced wing) configuration comes with compressive loads along unconventional paths. These compressive loads provide the possibility of static structural instabilities related to global buckling with an aeroelastic component. In Reference [9], it is clear that non-linear mechanics and follower forces contribute to the successful design of a joined-wing SensorCraft concept.

In Reference [13], flutter calculations were performed about a structurally non-linear equilibrium condition. In Reference [14], an unstructured Navier-Stokes solver is used to converge on a static aeroelastic equilibrium condition of a joined-wing HALE configuration.

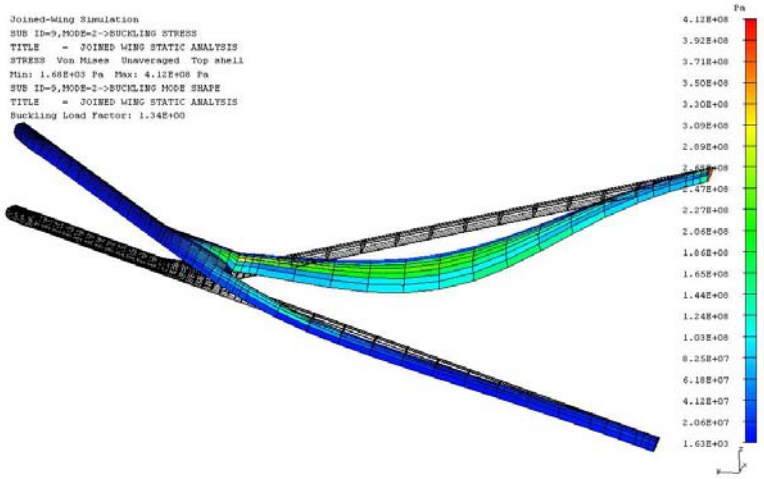


Figure 2: Critical Buckling of the Fully-Stressed Non-Linear FEM

Figure 2 comes from Reference [9] and is based on a built up membrane FEM model. Of course no membrane is unsupported, thus precluding the possibility of local panel buckling in the non-

linear analysis. Figure 2 depicts the critical buckling mode for an optimized structure with trimmed aeroelastic follower forces. Note, while the aft wing is shown in the buckled state, it would be more accurate to say the leading edge of the aft wing is buckled. During the optimization process, forward wing buckling is also a possibility along the leading edge for upward loads and along the trailing edges for downward loads.

Reference [15] covers the development of an equivalent plate model (EPM) for practical non-linear structural analysis at the preliminary level. The EPM avoids bothersome local panel buckling (detailed analysis) while establishing global load paths at the preliminary design level. However, this advantage is partially offset with the loss of detailed sub-structural modeling.

3.2 Technologies:

Numerous technologies apply to SensorCraft. These include advanced structural concepts, multifunctional structures, boundary layer control and various aeroelastically enhanced control effectors. The optimal design of a conformal load-bearing antenna structure was addressed in Reference [12 Smallwood]. The optimal performance of each of these technologies requires an integrated system perspective. For instance, all of the technologies listed effect sensor (antenna) performance. All these technologies affect the aeroelastic response of the overall system.

Thus, it stands to reason that a technology assessment of any one technology requires a comprehensive computational design model of the entire system that incorporates every technology. From a software perspective, this is a daunting long-term challenge. The software solution presented in this paper reaches out to the broadest possible user community with open-source development based on standardized compiled source code. This offers the ability to formulate computationally intensive design modules in C++ that can be readily integrated and optimized at the system level.

Figure 3:

Straight Wing [5]

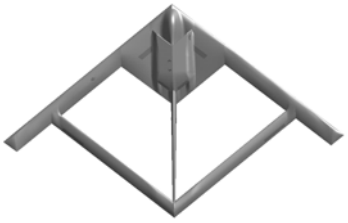


Figure 4:

Joined-Wing [1]

Figure 5:

Blended-Wing-Body [5]

3.3 Managing Configuration Design Variants

SensorCraft technologies are being evaluated (by major US airframe manufacturers) in the context of three basic configuration types. These configuration studies are valuable in putting technology development into a system context. However, one is challenged to objectively identify which configuration is best. This requires a comprehensive study involving data from many assessments of many design variants driven by a myriad of variables – a computational design study.

Industry designers are developing proprietary computational design capabilities. Examples were described in References [7] and [16]. Yet research opportunities are still ripe for the taking.

AVEC is a pilot for a non-proprietary research tool that will lead to the desired Computational Design capability.

An optimal design depends on the size of the design space and the fidelity (credibility) of the models. More design variants, technologies and fidelity reduce the chance that a less-than-optimal design will go into development. Of course, perfect and absolute optimization requires unconstrained and infinite possibilities. Thus, useful design optimization first requires human intervention to define a practical process using a combination of intuition and expertise.

The joined-wing concept in Figure 4 is especially noteworthy because of the unusual load paths discussed above. With geometric non-linear mechanics and follower forces in play, a practical aeroelastic design optimization process for the joined-wing concept is absolutely required. While Reference [9] was successful in completing a design optimization process for one configuration, the design convergence process was very cumbersome and not practical to address an entire family of designs. Reference [17] was successful in generating a response surface for a family of 70 design configurations. Each of the 70 structurally optimized configurations was based on buckling criteria. However insufficient time and resources were available to closely examine each of the 70 designs. The emphasis was on identifying a process for optimizing the response surface.

None of these joined-wing configurations closely matched the SensorCraft configuration depicted in Figure 4. So, we still struggle to develop computational design as a means to optimize a family of configurations with non-linear mechanics. This is the immediate goal of the AVEC development.

Table 1 and Figure 6 work together to describe a fairly comprehensive preliminary design process for any one SensorCraft design optimization. Table 1 lists a number of indexed data types that are passed about as a design process iterates. These indices are encircled in Figure 6 and make connections between output from one analysis module and input for another. For example (C) Parametric Geometry feeds (6) Outer Mold Surface for Component Layout which in turn is input for analysis module for (D) Component Layout. Waterfall diagrams such as this are a challenge to follow due to their inherent complexity. However, any design study that has not been traced with a flow diagram is probably not manageable. On the other hand, a waterfall diagram lacks significant detail which must be captured at the software documentation level.

1 Altitude, Mach
2 Altitude, Mach
3 Altitude, Mach
4 Required Thrust
5 Fuel Consumption Rate
6 Outer Mold Surface for Component Layout
7 Watertight Mold Surface for Aero Mesh
8 Surface and Structure Geometry for Mesh
9 Power System Position
10 Component Geometry
11 Component Masses
12 Aerodynamic Computational Mesh
13 Pressure Distribution for Structural Integration
14 Pressures for Drag and Range Calculations
15 Aerodynamic Sensitivities
16 Aerodynamic Loads for Structural Analysis
17 Fuel Consumed
18 Trimmed Fuel Status
19 Control Surface Settings
20 Adjustable Mass Locations (e.g. fuel mgt)
21 Control Effecter Settings
22 Vehicle Attitude
23 Total Vehicle Inertial Properties
24 All Masses for Structural Analysis
25 Structural Computational Mesh
26 Structural Deformations
27 Structural Element Stresses
28 Eigenshapes
29 Sensitivities to Flutter, Buckling
30 Structural Element Masses
31 Structural Element Thickness

Table 1: Design Procedure Indices for Figure 6

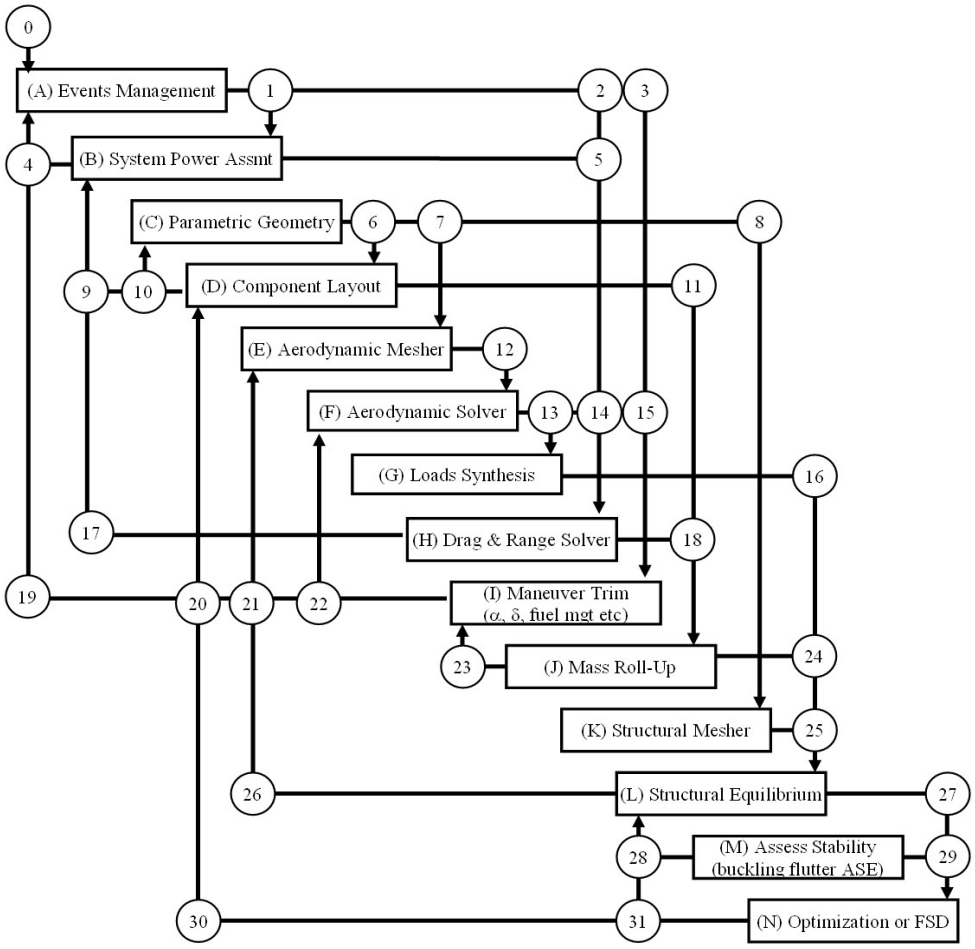


Figure 6: Design Procedure with Indices in Table 1

4. Overview of AVEC

Several requirements were imposed at the start of the AVEC pilot development. First, the code is to be *platform independent*. This meant we could use any software language that was available on all major operating systems including Unix, Linux, Mac and Windows. Second, we restricted the project to *compiled code*. This means the data variables are strongly typed. This will facilitate (in some way) error management, scalability and compatibility with database managers associated with optimization of computational physics. Third, we require the compiled computational design capability to readily adapt to either master or slave status. These three requirements restrict AVEC to two choices of compiled source languages, Java and C++. ANSI C++ was chosen based on personal familiarity, a choice I am sure will be punished. However, my expectation is that the same functionality will work just as well in Java as long as C++ pointers can find their equivalent replacement in Java. Aside from global declarations (unit conversions and file paths) all of AVEC falls under a root C++ class that can be instantiated and operated as part of other software developments.

As a side issue, scripted freeware languages (Python, Ruby) are gaining in popularity as the basis for managing design environments. This is discussed in Reference [1]. I believe the choice between compiled and scripted languages will remain a struggle for years to come. I have a sense that the scripted languages enable fast prototyping, but the compiled languages will find ways to replicate this functionality. Indeed, this is already taking place.

The work presented here benefited tremendously with TrollTech QT GUI builder, which also serves as an interface with OpenGL graphical rendering. This is a multi-platform library of utilities that is available under rules for shareware or commercially for proprietary development. Both avenues were used in the AVEC project.

AVEC will be an object-oriented environment with three levels of abstraction (i.e. levels of interaction). These are graphically depicted in Table 2.

Level	ABSTRACTION	VALUE
I	End User: Works interactively (no C++) • AVEC GUI: add, copy, modify, analyze • Link inter-object variables in dependency trail • Create/archive "super-classes" in XML format • Save and restore models with XML format	Tech Assessment • Mission Effectiveness • Materials Evaluation • Concept Optimization
II	Programmer: Engineer proficient in standard C++ • Component class derivatives • Construct virtual functions • Leverage geometry kernel	Configuration Innovations • Joined-Wing • Micro Air Vehicles • Morphing Air Vehicles
III	Software Specialist: expert in C++ • GUI: New interfaces • Develop new OpenGL features • Mesh and Analysis Classes • Optimization Methods	Design Methods Research • Non-Linear Physics • Coupled Physics • Uncertainty Mgmt • Distributed Design

Table 2: Three Levels of AVEC Abstraction

4.1 [Level I: End User] Most design engineers will understand AVEC as a graphically interactive design tool with save and retrieve functions in the form of XML file output and input.

The end user will appreciate the ability to develop new parametrically-driven models from compiled classes.

4.2 [Level II: Programmer] Object-Oriented programmers will immediately appreciate the ability to inherit classes from the AVEC library to create new classes. For instance, with some C++ knowledge, new geometric entities should be fairly painless to add. Indeed, AVEC anticipates the adoption of a standard geometry kernel in the not-too-distant future. This would extend the simple set of geometric entities (line, curve, surface etc) already available in AVEC pilot code.

4.3 [Level III: AVEC Collaborator] AVEC collaborators in the form of hard-core C++ programmers (not that I consider myself deserving of this title) may be interested to look into the AVEC source code and either make modifications to existing functionality or enhance current capability. For instance, AVEC is well suited for development of cascaded uncertainty along with the current dependency-management system. From an engineering perspective, this could be applied in numerous situations. For example, this means the design might be extended beyond classical structural reliability-based design to include uncertain loads.

Levels I and II are discussed in Section 5 and Section 6 respectively.

5. End-User Functionality of AVEC (Level I)

The End User will interact with the AVEC graphical interface to develop models and families of models that automatically benefit from native AVEC functionality in the form of dependency-management. These models and family of models can be saved and retrieved in XML format. The expectation is that AVEC data that conforms to an XML schema can be viewed with an XML browser. Numerous on-line documents[1] [2] [3] are available that cover everything about XML schemas. Examples XML browsers[4] are also available on-line. Ultimately, AVEC needs a browser that will examine its XML model file and generate a UML flow diagram of its instantiated model. This will be conceivably possible because data variables and functions are wrapped independently. Data dependency is contained as part of AVEC's XML data structure. Variable values are saved for independent variables only. Slaved values point to their master variable. Proposed/Example XML data format for model save and retrieve operations are provided in Figure 7.

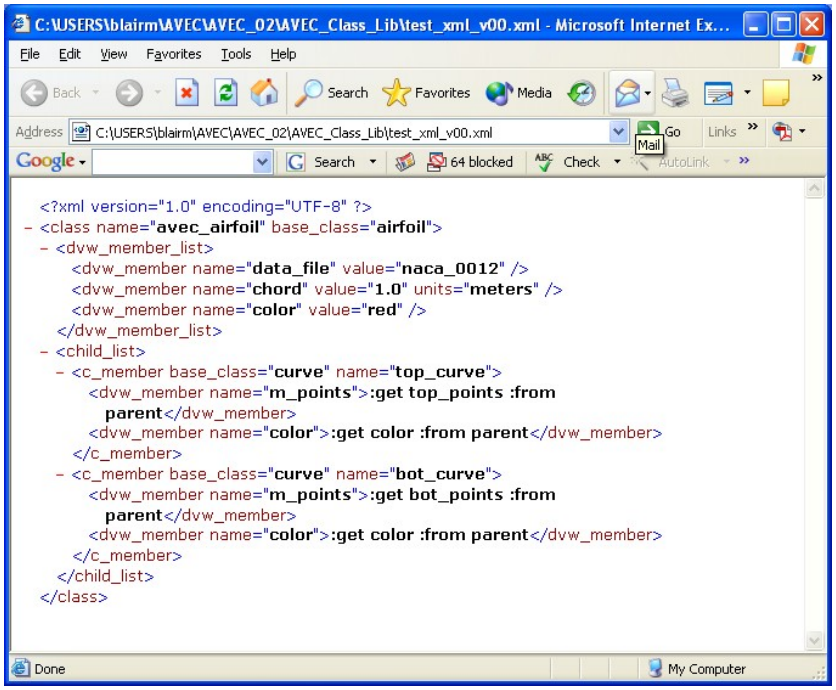


Figure 7: Sample XML Data Format

[1] [http://www.w3.org/XML/Schema], ,
[2] [http://www.xfront.com/]
[3] [http://www.w3schools.com/schema/default.asp]
[4] [http://www.xml.com/pub/rg/85]

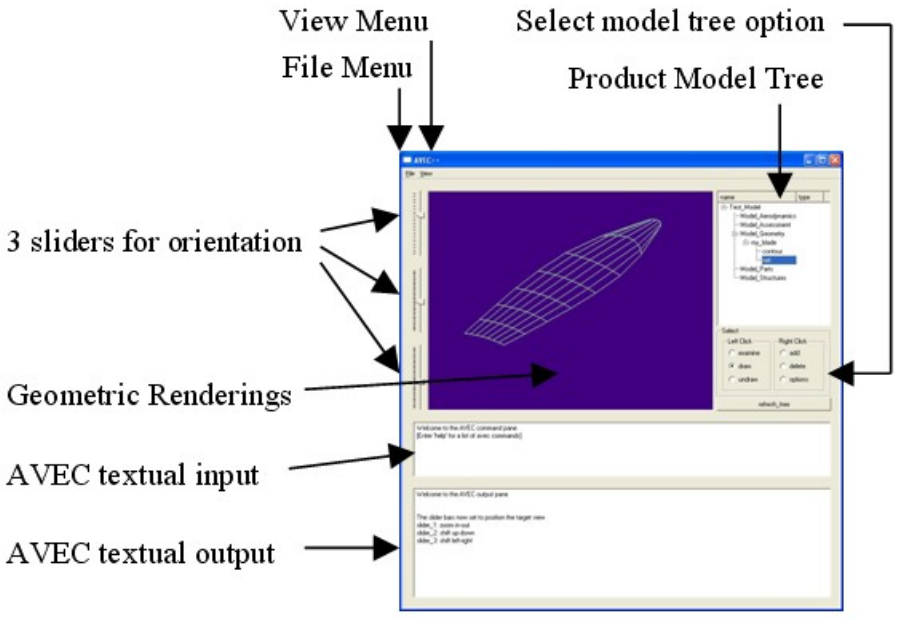


Figure 8: GUI-view of AVEC for Interactive End User – Main Pallette

Figure 8 highlights the various features of the main palette. The main viewing window contains graphical renderings. The FILE menu offers save and retrieve functions. This provides the ability to save interactive model developments on disk storage and retrieve them for later review and enhancements. The VIEW menu offers options for various AVEC viewing functions on the graphical rendering in the main viewing window. Viewing control is affected through three sliders bars. Thus, the sliders control viewing translation, rotation, zoom and more. The product model tree and associated radio buttons (see Figure 8) work in tandem. The product tree presents a design model in hierarchical (parent/child) form. The radio buttons provide the end user the ability to add, modify, render, delete, save and retrieve components in the product tree. Textual input and output are contained in the bottom horizontal boxes.

In addition to the ability to save and retrieve entire models, the end user is able to save composite classes (partial models) that can be reused along with any other AVEC native classes. For instance, the end user can interactively construct and save an airfoil class that can be interactively retrieved as part of user defined wing and tail classes. Data is saved in XML format, the same format as used to save an AVEC model. However, variable values are not saved for derived classes. A number of derived classes will be part of the distribution following the pilot code development.

Figure 8 depicts a partial graphical view of the AVEC desktop as implemented today. The graphically rendered Micro Wing is displayed as part of the derived Blade class and children. The instantiated object tree is depicted to the right of the graphical rendering. Below the object tree is a palette of options that control the action upon selecting any one object listed in the object tree. The textual boxes at the bottom facilitate model modifications with text input and output. An AVEC help utility is suitable for such user-driven I/O.

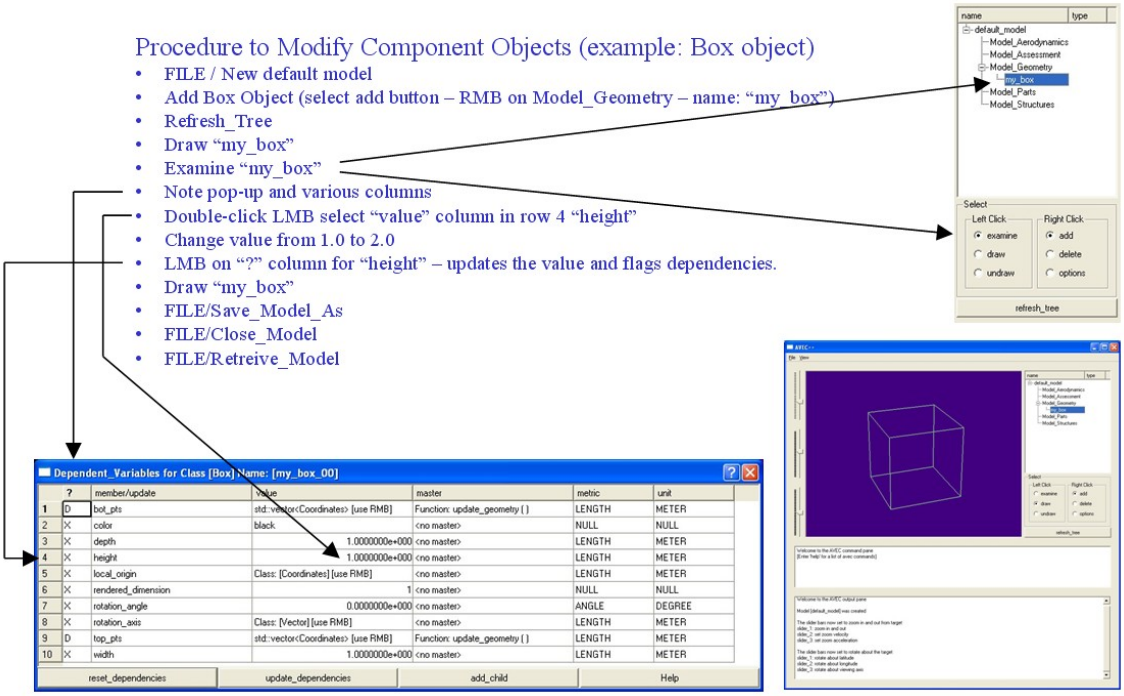


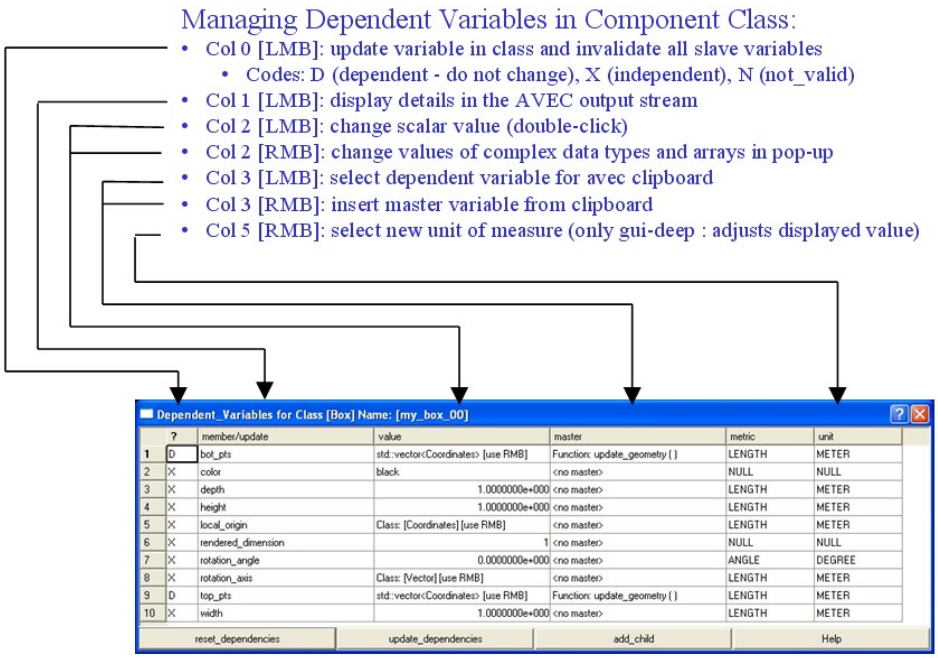
Figure 9: GUI-view of AVEC for Interactive End User – Examine Class

Figure 9 introduces the interactive process whereby the user instantiates an AVEC class (Component) and subsequently makes changes to member variables (i.e. wrapped in class Dependent_Variable_Wrappers) within model objects. Three panes are displayed in Figure 9. The main pallette is displayed in the lower right corner. The upper right portion of the main pallette contains the model tree and associated action buttons (ie. radio buttons). The radio buttons determine actions for the left and right mouse buttons. This model management box is magnified in the top right corner of Figure 9. A grahical pane with a list of variables is displayed in the lower left corner of Figure 9.

Interactive user actions are listed in Figure 9 and arrows point to the relevant display window. The first action refers to the FILE menu option in the top toolbar. The next action directs the user to select "New default model" from the FILE menu. This will generate a default model tree to appear in the model management box. With radio button "add" selected, next right click on "Model "Geometry" (in the model tree in the model management box) and add object "my_box". Select button "refresh tree" in the model management box and "my_box" will appear in the model tree. Next, with radio button "draw" selected, left click on "my_box". The image of a box will appear in the pane for geometric renderings. With radio button "examine" selected, left click on "my_box".

A pane will pop up with a list of all the pre-determined (by the programmer, see Section 6) dependent variables listed, as displayed in the lower left corner of Figure 9. Each variable can be managed here in terms of its value, master variable, and unit of measure. For variables that represent vectors and matrices of native variables and AVEC_Datatypes, a pop-up text window provides a mechanism for entering variable values. Additional details are provide with the aid of Figure 10. The remaining actions listed in Figure 9 are somewhat self-explanatory for this top-

level explanation of AVEC end user functionality. When variable values are reset, graphical renderings disappear and can be redrawn.



	?	member/update	value	master	metric	unit
1	D	bot_pts	std::vector<Coordinates> [use RMB]	Function: update_geometry ( )	LENGTH	METER
2	X	color	black	<no master>	NULL	NULL
3	X	depth	1.0000000e+000	<no master>	LENGTH	METER
4	X	height	1.0000000e+000	<no master>	LENGTH	METER
5	X	local_origin	Class: [Coordinates] [use RMB]	<no master>	LENGTH	METER
6	X	rendered_dimension	1	<no master>	NULL	NULL
7	X	rotation_angle	0.0000000e+000	<no master>	ANGLE	DEGREE
8	X	rotation_axis	Class: [Vector] [use RMB]	<no master>	LENGTH	METER
9	D	top_pts	std::vector<Coordinates> [use RMB]	Function: update_geometry ( )	LENGTH	METER
10	X	width	1.0000000e+000	<no master>	LENGTH	METER

Figure 10. Examine Component Pane

The widget (table) in Figure 10 lists all the dependent variables contained in an instantiated Box object (Box class inherits from Component class). For instance, variable values, “height”, “width” and “depth” control the box linear dimension and “rendered_dimension” controls the graphical rendering with 0 (points), 1 (wire frame) and 2 (surface) dimensions. This widget is displayed when the examine button (see Figure 9) is selected and an item (my_box) in the object tree is selected. The left column (Col 0) in the table widget identifies the status (D for dependent, X for Independent and Valid, N for Independent and Not_Valid). The second column (Col 1) lists the variable name. The third column (Col 2) displays the data value or the data type for large vectors and arrays. A right-click on a box in the third column brings up a dialogue box with all values listed. If the variable status is Independent, then the values can be modified, appended or deleted. Column 3 is dedicated to identifying and modifying dependency on a master variable or just identifying its master function. (Variables and functions are wrapped independently). Column 4 is static and identifies the metric type. Column 5 is variable and identifies the unit of measure. The user can select cells with the right-mouse-button in this column and subsequently select from a list of available units of measure. In AVEC, the unit of measure displayed determines the scaling value on the variable value that is entered in the value column. All values are stored in terms of the base units of measure.

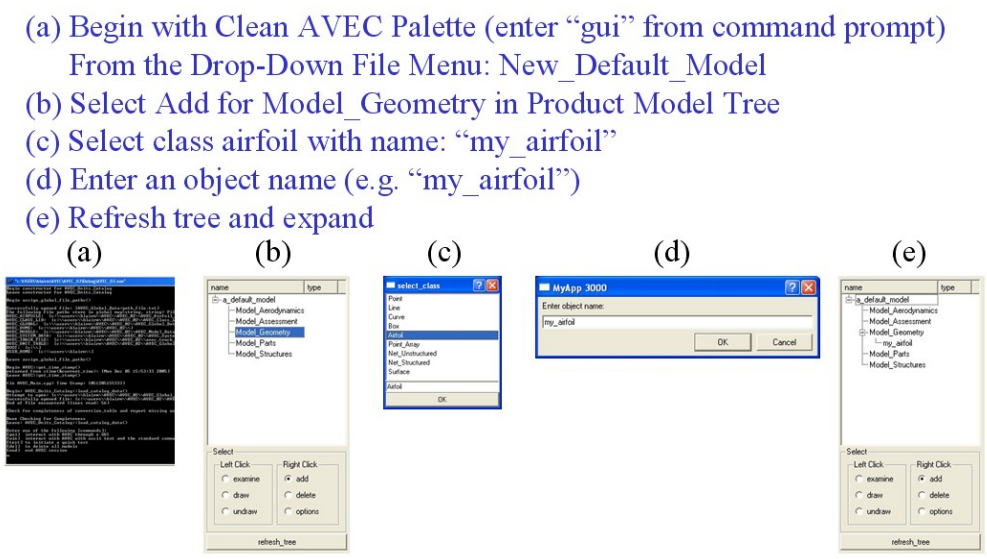


Figure 11a. Derived Airfoil Class

Figures 11a through 11d describes the process for creating a derived class from a number of compiled AVEC classes. This process is described in terms of an airfoil example. By itself, an airfoil class does not generate graphical rendering. Two children of class "curve" are appended to the instantiated airfoil object to render the top and bottom curves of an airfoil. Children of class "point_array" are appended to each curve, thus providing a simple mesh of points for analysis. All together, this collection of objects form a derived class. Derived classes are saved in XML format and can be retrieved. Variable values are not saved in derived classes. This process is described as follows.

Figure 11a depicts a series of five windows labeled (a) through (e) associated with steps (a) through (e) that describe the process for instantiating a new model from a compiled AVEC class. The AVEC classes are developed by programmers and stored in any number of library collections as described in Section 6.

Window (a) in Figure 11a depicts the Windows XP Command Prompt from which AVEC can be started for textual control. The textual interface for AVEC is not yet well developed, but will become necessary for remote batch applications and/or as an instantiated class within another larger compiled parent project. In the example below, the AVEC main pallete is instantiated from the Windows XP Command Prompt.

By step (e) in Figure 11a, an AVEC model has been instantiated with airfoil class intantiated.

(f) Select Add for "my_airfoil" in Product Model Tree
(g) Select class "Curve" with name: "top_curve"
Refresh and Expand Tree
Repeat (a) and (b) for "bot_curve"
(h) Select Add for "top_curve" in Product Model Tree
(i) Select class "Point_Array" with name: "mesh"
Refresh and Expand Tree
(j) Final Tree after including "mesh" under "bot_curve"

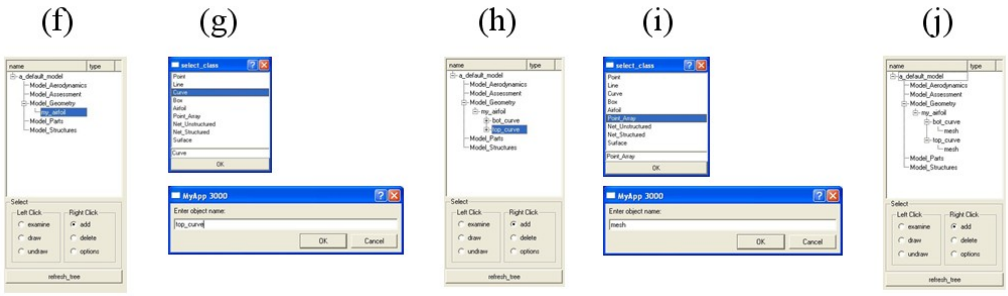


Figure 11b. Derived Airfoil Class – Adding Children

Figure 11b depicts the process for adding two children to airfoil object. These children are of class curve. These curves will graphically render the top and bottom curves of the airfoil. Subsequently, children of class "point_array" are added to each of the curve objects. Steps (f) through (j) in Figure 11b follow step (e) in Figure 11a.

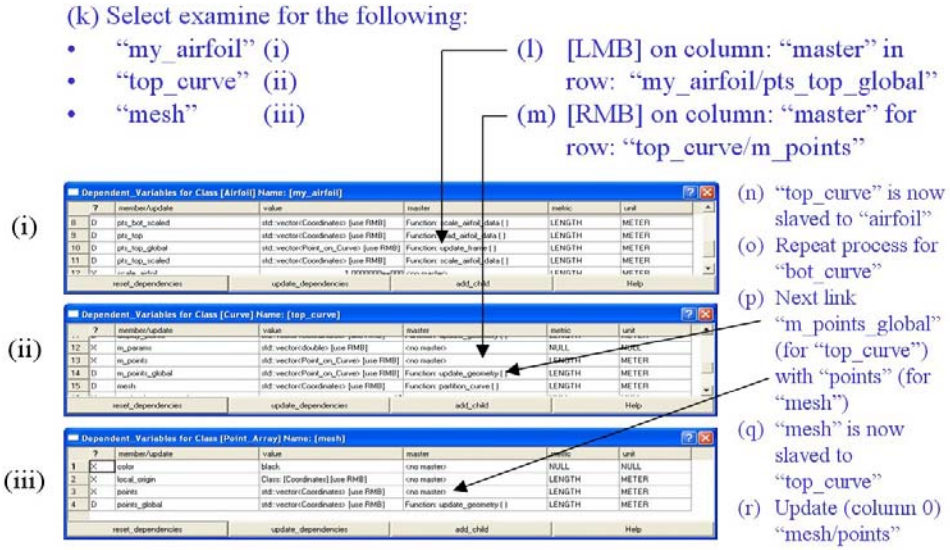
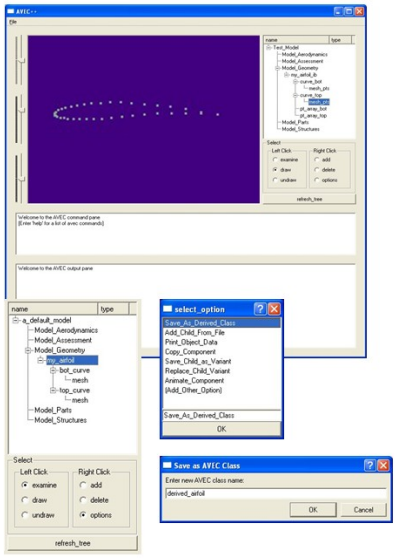


Figure 11c. Derived Airfoil Class – Establish Master/Slave Dependencies

With classes intantiated, the member variables need to be interactively connected in master/slave relationships. This interactive process is described in Figure 11c in terms of steps (k) through (r). Step (k) brings up the variables in a separate window for each of the instantiated classes.

With LMB in step (l) the variable pts_top_global in class airfoil are selected to serve as master. Subsequently, with RMB in step (m), the variable m_points in "top_curve" are selected to serve as slave to "pts_top_global" in class airfoil. The procedure is repeated in the following steps to link variables in the top and bottom curves and their children meshes of class "point_array".

Figure 11d completes the process with steps (s) through (w). The airfoil is drawn in terms of the top and bottom mesh points. The airfoil can also be drawn in terms of the top and bottom curves. At this point, the composite of classes can be interactively saved and subsequently retrieved as a derived class. The derived class will retain all dependent functionality. Thus, the coordinates of various airfoils can be retrieved from a library of text files with airfoil coordinates and the resulting airfoils can be placed oriented and rendered to become part of a geometric wing assembly.



s) Select "draw" and "mesh" (see figure)
t) Select "option" and "my_airfoil".
Pick: "Save_as_Derived_Class"
u) Select FILE/Delete_Model
v) Select FILE/New_Default_Model
w) Select "option" and
"Add_Derived_Class_As_Child"

Saved XML File Format (abridged)

```
<?xml version="1.0" encoding="UTF-8"?>
<class name="avec_airfoil" base_class="airfoil">
    <dvw_member_list>
        <dvw_member name="data_file" value="naca_0012"> </dvw_member>
        <dvw_member name="chord" value="1.0" units="meters"> </dvw_member>
        <dvw_member name="color" value="red"> </dvw_member>
    </member_list>
    <child_list>
        <c_member base_class="curve" name="top_curve">
            <dvw_member name="m_points"> :get top_points :from parent </dvw_member>
            <dvw_member_name="color"> :get color :from parent </dvw_member>
        </c_member>
        <c_member base_class="curve" name="bot_curve">
            <dvw_member name="m_points"> :get bot_points :from parent </dvw_member>
            <dvw_member_name="color"> :get color :from parent </dvw_member>
        </c_member>
    </child_list>
</class>
```

Figure 11d. Derived Airfoil Class – Save and Retrieve

In reality, AVEC is a C++ programming environment from which any number of end uses could be served. The above description serves as a simple example of an end use. Each end user developed by a C++ programmer will require a Users Guide before distribution. The following section describes AVEC from the programmer perspective at a high level. Again, a complete Programmer's Manual will be required before AVEC is distributed.

6. Software/Programmer Functionality of AVEC (Level II)

End users are warned that this section contains details of object-oriented source-code. However C++ (and maybe Java) programmers will be familiar with the concepts used in this section. The information presented here is in the format of a overview. In addition, some effort has gone into on-line documentation[5] of AVEC. This interactive programmer's document (in HTML) is used as a reference manual.

Computer programming is pure abstraction, ultimately taking the form of binary numbers that represent either data or instructions (i.e. procedures in the larger sense). Computer languages were developed in order to bridge the gap between human language and binary instructions. Still computer languages are only slightly less abstract. The syntax of computer languages has little to do with human experience. Early languages (e.g. Fortran) were developed to solve large algebraic (also referred to as computational) problems. Computer languages have evolved. Now, engineers with programming knowledge are able (perhaps inspired) to use computer programming languages to organize data and procedures and thereby produce graphical renderings that an end user (e.g. engineering designer) can use in a simulation that mimics human experience. AVEC combines computational functionality and graphical renderings into a single programming environment for computational design. AVEC facilitates programmers in developing computational design applications for engineering designers. An example was discussed in Sections 2 and 3.

Object-oriented programming (OOP) has become a popular form of computer programming. OOP has enabled teams of programmers to collaborate on single projects of tremendous complexity represented perhaps by the entire computer gaming industry. OOP is also the basis for most engineering analysis software that is commercially active.

OOP builds on traditional procedural programming languages (e.g. the C programming languages) with class structures. A class structure is a combination of data and procedures encapsulated into a single entity. The class structure is programmed or organized in terms of a computer language (e.g. C++). Any number of classes can be inserted (i.e. declared) within a computer procedure (again C++ is also a procedure like C). When a program is running with class structures, the classes take on a "life" and the class is said to be instantiated. A class can be design to encapsulate (contain) any number of data variables, or procedures. A class can be designed by a programmer to contain any number of instantiated classes, or the programmer can declare an open-ended list of objects that the end user has instantiated.

AVEC is a single class structure programmed in C++. Once instantiated, the AVEC user fills the AVEC class with a number of instantiated objects in organized lists. For instance, a computational design model (e.g. an airplane design model) is contained in a list that contains objects of type "Component". The AVEC program assigns these component objects access to other component objects in order to form an abstract tree of parent-child relationships.

OOP programmers can use class inheritance to develop specialized classes from a single base class. For instance, the Airfoil object discussed in Section 5 is instantiated from an Airfoil class. The Airfoil class is source code that inherits its base functionality from class Component. Airfoil class can interactively interact with class Curve because they both inherit from class Component.

[5] DOxygen is downloadable freeware www.doxygen.org for automated on-line documentation generation.

These various OOP abstractions are thoroughly addressed in many excellent textbooks in print. But, still, these programming abstractions are a necessary challenge to communicate to an engineering community seeking a better computational design environment.

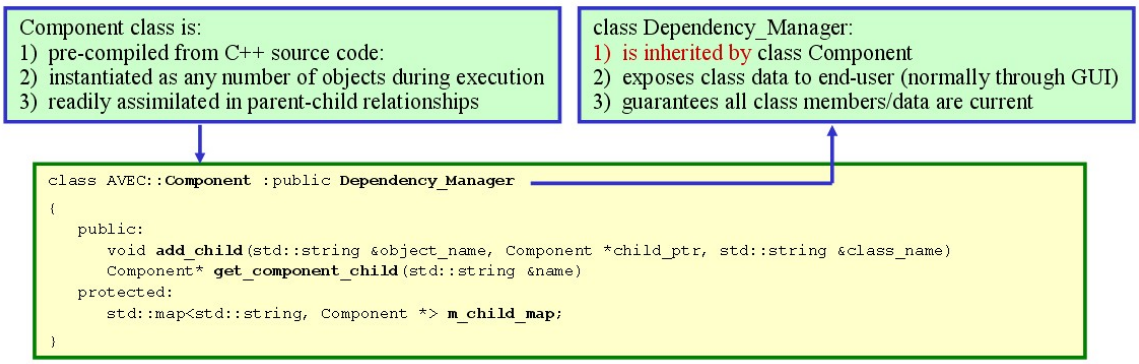


Figure 12a: AVEC Class Inheritance

Figures 12(a-c) and Figure 13 exemplify programming abstractions. These figures are formatted in tree-like structures. But Figures 12(a-c) represent something very different from Figure 13. Figures 12(a-c) describes class inheritance (classes that inherit are derived from other classes during compilation). Figure 13 addresses the issue of parent-child hierarchical relationships for instantiated objects. Class inheritance at the programming level and parent-child relations instantiated during run time was discussed above. Likewise, the distinction between Figures 12(a-c) and Figure 13 is brought out in the following paragraphs.

6.1 Class Inheritance in AVEC

Figures 12(a-c) provide a graphical perspective to AVEC Component class inheritance. This graphical representation is significantly truncated to the level of a cartoon rendering. Consistent with all object-oriented programming, new derivative classes can be created by building on (i.e. inheriting from) existing classes. Classes that inherit from Component class will automatically have the ability to relate to other instantiated Components in parent-child relationships.

As indicated in Figure 12a, Component class inherits from Dependency_Manager. Dependency_Manager enhances Component class with the ability to declare and manage any set of variables with persistence. Here, persistence means that any changes in the data will be carried through the entire model according to a customized (user-defined) dependency trail. Persistence is managed efficiently. The valid/invalid Boolean status persists. But time-consuming calculations are delayed until the user requests dependent data that was earlier declared to be invalid. This feature is called “lazy evaluation” or “demand-driven” in the literature.

Dependency management is an effective tool when properly integrated with a design process. The cost of dependent data storage should be balanced with the cost of dependent data recalculation. Each wrapped variable (or function) comes with significant data overhead to manage its Boolean valid/invalid status and maintain all slave/master dependency declarations. AVEC programmers (Level II) have the freedom to select key variables for the dependency trail and secondary variables buried in member functions.

So, how does dependency management facilitate a computationally intensive design process? After years of experience, the answer is not as obvious as once thought. The envisioned AVEC system anticipates the interactive construction of large families of design variants that compete to serve a common mission. New members of a product family will be created with native object copy capability. New member copies are subsequently distinguished from its progenitor with a modified set of parametrics. For the parameters that are not changed, the automated dependency management is maintained as part of the copy process. One will also use dependency management to control a string of lengthy analysis and operations. The dependency manager will require significant enhancements before addressing design cycles in an automated convergent optimization procedure. Also, in future developments, the existing dependency management class could be enhanced to address and manage cascading uncertainties. These are opportunities for research, the raison d'etre for AVEC.

Variable values in AVEC can be transferred in terms of its equivalent string value or shared in terms of its pointer address in memory. The ability to exchange an equivalent string value enables future expansion of AVEC to include geographically distributed modeling (multiple computers). The ability to share pointer memory is a major convenience where software integration is concerned. For instance, shared pointer memory saves the pain of translating each data item when an entire class is passed.

Also note in Figure 12a, a small portion of the class contents are shown. These include two class functions, add_child( ) and get_component_child( ). The one data item listed is child_map. Member variable child_map is of type std::map. The notation "std" is source code shorthand for the C++ Standard Template Library (STL) – a recent addition to C++. Among other features, STL significantly facilitates data array/vector management (for ANY data type or class) and map is a member of this library. The std::map class is essentially a vector in which the programmer can reference individual members in terms of a key character string instead of an integer index.

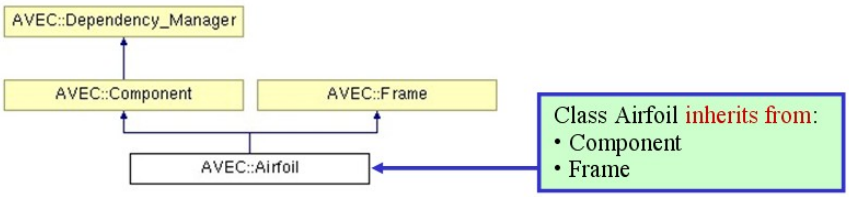


Figure 12b: Inherited Classes Contained in AVEC Class Airfoil

Two examples of class inheritance are shown in Figure 12b and 12c. Class Airfoil in Figure 12b inherits from both Component and Frame classes. With Component, Airfoil is automatically prepared to be part of a larger Component parent-child data tree. With Frame, Airfoil coordinates can be rotated and translated such that it lines up with other Components. Class Box is a second example of class inheritance in Figure 12c. As with Airfoil, Box also inherits from Component and Frame classes. Box also inherits from class Display_List_Member. Display_List_Members can be tailored to drive OpenGL graphical rendering. Thus, Box object can be drawn. In contrast, Airfoil alone cannot be drawn. In order to draw Airfoil, other child classes will be called on as indicated in Section 6.3.

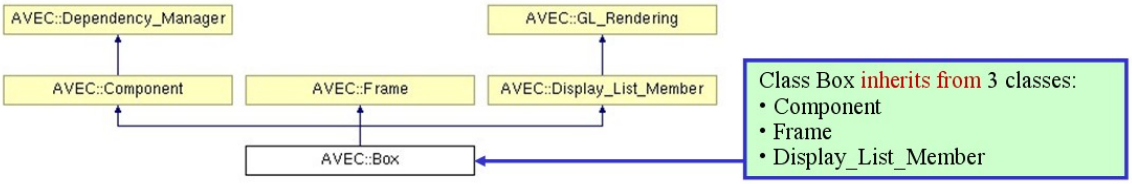


Figure 12c: Inherited Classes Contained in AVEC Class Box

6.2 Programming with Virtual Functions in AVEC:

Virtual functions are a standard part of C++. Virtual functions are members of a class that can be specialized to support inherited classes. Table 3 provides a list of virtual functions and their parent class.

FUNCTION NAME	SUPPORTS	PARENT CLASS
• update_geometry( )	geometry kernel	Display_List_Member
• update_display_queue( )	GL graphics rendering	Display_List_Member
• update_frame_coordinates( )	geometry orientation	Frame
• install_dependent_variables( )	declare dependent variables	Dependency_Manager
• install_dependent_functions( )	declare dependent functions	Dependency_Manager

Table 3: Virtual Functions in AVEC

Now we can explain how a virtual function facilitates the drawing of items in the product model tree in Figure 8. Each item in the product model tree is of class (or inherits from class) Model_Tree_Item. Model_Tree_Item contains a pointer to class Component or a variant of Component such as Box or Surface etc. [e.g. Class Box (Figure 12c) inherits from Component]. Box also inherits from Display_List_Member (the order of inheritance is critical). So how does a Model_Tree_Item know how to distinguish between its variants? In other words, how does Model_Tree_Item know when to draw a Box and when to draw a Surface?

In Table 3, we see that update_display_queue( ) is a virtual function in Display_List_Member. Class Box (Figure 12c) inherits from Display_List_Member. A programmer created a specialized version of update_display_queue() within Box class. The specialized version creates OpenGL directives to draw a box using a simple set of calls. Now, Box::update_display_queue() is a particular virtual function that can be referenced as if it were a generic Display_List_Member:: update_display_queue( ). The various members of the product model tree in Figure 8 can be drawn as long as they point to component variants that also inherit from class Display_List_Member.

If this explanation seems a bit convoluted (and it is) and if this points out the complexity of C++ (and it does), then a C++ programmer (Level II) does not need to be concerned. AVEC has already taken care of the complexity and the programmer only needs to create a specialized virtual function update_display_queue() for any new variant of Component. Because AVEC will be open source, the programmer has access to many self-explaining examples such as the source for class Box. Also, we might remember the axiom: If you want to program with “powerful” and capable software, there is going to be complexity. The AVEC programmer (Level III) has

designed in C++ virtual functions to make the C++ programmer's job (Level II) easy. Subsequently, the C++ programmer's job is to make AVEC easy for the end user (Level I) to operate at the CAD level.

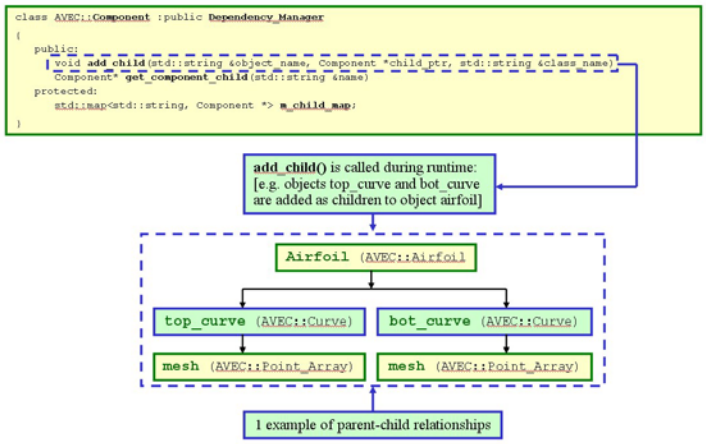


Figure 13: AVEC Instantiated Object Hierarchy

6.3 Object Hierarchy in AVEC:

Figure 13 depicts the AVEC capability to instantiate (at run-time) Component children. The example is Airfoil class with two children and two grandchildren. The two children are Curve class that allows the airfoil to be discretized and graphically rendered. Each Curve class has one child that allows the curve discretization to be graphically rendered. While Airfoil is a native AVEC class, Airfoil alone cannot be drawn (graphically rendered). The Airfoil assembly depicted in Figure 13 is an end user-defined (derived) class that can be saved-in and recalled-from a library of user defined classes.

6.4 Dependency Management in AVEC:

In Figure 12a, we saw that class Component inherits from Dependency_Manager. Dependency management guarantees that every piece of model data is consistent with respect to its master variable. Figure 14 depicts some details of the Dependency_Manager class. The reduced version of Dependency_Manager shown contains two public (any user of this class has access to public data) functions and two protected (can only be accessed by class member functions) data items. These two data items, dfw_list and dvw_list, are of type std::map. The map dfw_list is essentially a vector of class type Dependent_Function_Wrapper. The map dvw_list is essentially another vector of type Dependent_Variable_Wrapper. A source description of Dependent_Function_Wrapper (DFW) and Dependent_Variable_Wrapper (DVW) are provided in the respective call-out boxes in Figure 14.

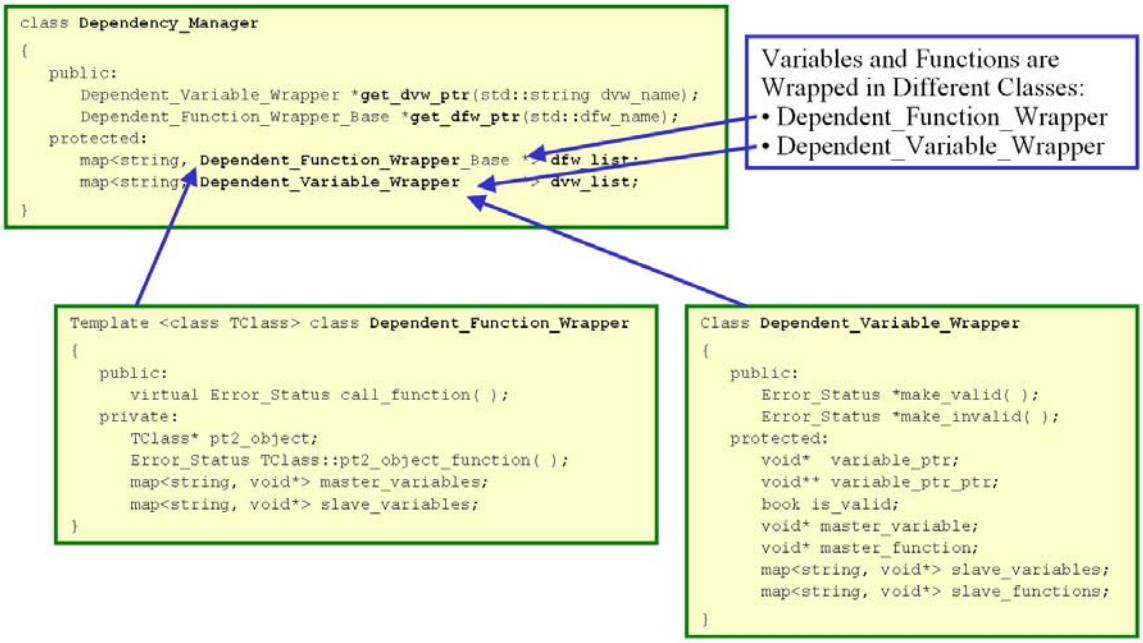


Figure 14: AVEC Dependency Manager

Dependent_Variable_Wrapper class contains seven protected member variables. The variable type void* is peculiar to C++. The * is an indicator that the named variable is to be managed in terms of its position in memory. The * syntax is a declaration for a C++ pointer. The type void is a catch-all data type. The void data type must be type-cast (e.g. as int or float etc) wherever void is operated upon. The type void** is a double pointer. Essentially, this is a pointer to a pointer of type void. C++ manuals warn of the dangers of void and pointers. However, with skillful care on the part of the AVEC programmer (Level III), the operation of Dependent_Variable_Wrapper can be (I want to say: has been) guaranteed to behave gracefully. In Figure 14, we see Dependent_Variable_Wrapper contains the following members:

- variable_ptr — Points to memory location for any type variable of interest. Exception is variable_ptr_ptr.
- variable_ptr_ptr — A pointer to pointer in memory and used to wrap pointers
- is_valid — Declares whether the value behind variable_ptr is consistent with master variables
- master_variable — NULL or pointer to the DVW that controls the value of variable_ptr.
- master_function — NULL or pointer to the DFW that controls the value of variable_ptr.
- slave_variables — STL map to a list of DVW whose Boolean value for is_valid is set to false when the present (in this context, the master) variable is changed and/or the present value of is_valid is set to false.
- slave_functions — STL map to a list of DFW whose variable_ptr is controlled by the present variable_ptr.

The two public member functions make_valid() and make_invalid() shown in Figure 14 are fairly complex procedures that cascade through an instantiated Component object tree (described above in the sense that class Component inherits from Dependency_Manager).

DFW class structure is a bit more complex (than DVW class). Some of the details are found in Reference [1]. For instance, one might be interested to read about "functors".

Classes that inherit from Dependency_Manager (and/or Component) have the option to declare virtual functions that declare all dependent variables and functions within the class.

- install_dependent_functions()
- install_dependent_variables()

The benefits of virtual functions in AVEC were discussed in Section 6.2 above. It involves a small amount of programming effort for C++ programmers (Level II) with the result that the class becomes very manageable for level one (end users) who can interactively declare dependencies between variables of different classes. See the example function declared in Figure 15 for Box::install_dependent_variables(). A careful observer can notice that the declared variables in Figure 15 (for class Box) are automatically displayed in Figure 10.

```
Error_Status Box::install_dependent_variables() //called in Component::add_child()]
{
Error_Status err;
string  str_name;
string  str_metric;
string  str_unit;
char dim_0 = 0;
char dim_1 = 1;

 str_metric   = "NULL"; str_unit = "NULL";
   str_name   = "color";
        err   = add_dvw_item(this, &color,    's', "string",           dim_0, str_name, str_metric, str_unit);
   str_name   = "rendered_dimension";
        err   = add_dvw_item(this, &rendered_dimension, 'i', "integer", dim_0, str_name, str_metric, str_unit);

 str_metric   = "LENGTH"; str_unit = "METER";
   str_name   = "height";
        err   = add_dvw_item(this, &height,   'd', "double",           dim_0, str_name, str_metric, str_unit);
   str_name   = "width";
        err   = add_dvw_item(this, &width,    'd', "double",           dim_0, str_name, str_metric, str_unit);
   str_name   = "depth";
        err   = add_dvw_item(this, &depth,    'd', "double",           dim_0, str_name, str_metric, str_unit);
   str_name   = "local_origin";
        err   = add_dvw_item(this, &local_origin,  'a', "Coordinates", dim_0, str_name, str_metric, str_unit);
   str_name   = "top_pts";
        err   = add_dvw_item(this, &top_pts,  'a', "Coordinates",      dim_1,  str_name, str_metric, str_unit);
   str_name   = "bot_pts";
        err   = add_dvw_item(this, &bot_pts,  'a', "Coordinates",      dim_1,  str_name, str_metric, str_unit);
   str_name   = "rotation_axis";
        err   = add_dvw_item(this, &rotation_axis,  'a', "Vector",     dim_0, str_name, str_metric, str_unit);

 str_metric   = "ANGLE"; str_unit = "DEGREE";
   str_name   = "rotation_angle";
        err   = add_dvw_item(this, &rotation_angle, 'd', "double",     dim_0, str_name, str_metric, str_unit);
return(err);
}
```

Figure 15: Virtual Function Box::install_dependent_variables

AVEC is set up to facilitate C++ programming of specialized Component classes (thus Dependency_Manager through inheritance) with a set of declared dependent variables and declared dependent functions that are compiled as part of the class. Dependencies between declared variables (DVW) and functions (DFW) WITHIN a class are also compiled as part of the class. In this way, the behavior of the class can be guaranteed by the programmer upon instantiation (loaded during run-time).

Following object instantiation, the end user (Level I) can establish dependencies between variables (DVW) of one object with compatible variables (DVW) of other objects. Recall the procedure for establishing inter-object dependencies was outlined in Figures 11c.

Dependency management is employed in AVEC with some implicit rules. A DVW can point to either one master variable or one master function. A DVW can point to many slave variables. A DFW can point to many master variables (DVW) and many slave variables (DVW). But a DFW cannot point to another DFW.

With this understanding of dependency management, we can return to Figure 6 with greater insight. The boxes along the diagonal are roughly representative of instantiated AVEC classes (yet to be developed of course). Each box in Figure 6 is labeled according to a class function. In AVEC, dependent functions (and associated dependent variables) are compiled as part of a class structure. The circles are representative of user defined links between dependent variables. The process for declaring these variable links was depicted in Figure 11c. Of course, as indicated in Figure 6, design processes are cyclical. That is, a design process iterates until it converges to produce desired results. At present, AVEC Dependency_Manager does not address the design convergence process. Some thought has gone into designing such a system. However, at present it remains a research topic. An open-source AVEC (or equivalent) will facilitate such research.

AVEC admittedly lacks much of the convenient and perhaps glitzy interactive features that are characteristic of the scripted languages identified in Section 4 and Reference 1. However, those features did not significantly contribute to the operation of the computational design process described in Reference 9. The scripted code used in Reference 9 was excellent for prototyping the process. However, the need for development of a design optimization process for a family of design variants is cause for pause and rethinking. AVEC is being designed with this experience in mind as well as the need for researchers to have access to far-reaching design optimization models of advanced systems.

As an open system, AVEC can become whatever an AVEC Programmer wishes. In this area of compiled vs scripted environment, it is not hard to imagine incorporating scripting functionality at the DVW class level within the larger compiled AVEC system. This feature would be appreciated at the end-user level who benefits with on-the-fly interactive functionality. It is also not hard to imagine other ways to enhance DVW class with uncertainty quantification for instance.

7. AVEC Pilot Status Update

As indicated, AVEC is a pilot code in development. The purpose of a pilot code is to create solid requirements for production code. This purpose is being achieved. To date, most attention has gone into development of geometric entities. In Figure 6, this relates to block C Parametric Geometry. The status of recently added or enhanced features is described here:

7.1 Units of Measurement

The management of units of measure is a feature of AVEC. In AVEC, units of measure are user specified. The conversion data for the global table is archived in an ASCII file that is easily modified by the end user (Level I). Class AVEC_Units_Catalog is declared in global space as avec_units_catalog. Conversion data for this class is read from file in the main( ) calling program. Data is automatically loaded when AVEC is instantiated. Base units of measurement are maintained in any instantiated class of type Model. Units are converted to and from base units only when the end user interacts with class variables for data viewing and modification (normally through a graphical user interface). Thus, the model is guaranteed to use a consistent set of base units (e.g. meters for length, kilograms for mass, seconds for time, etc). Units of measurement at variable level are user specified and arbitrary to the extent unit conversion is declared in the global table.

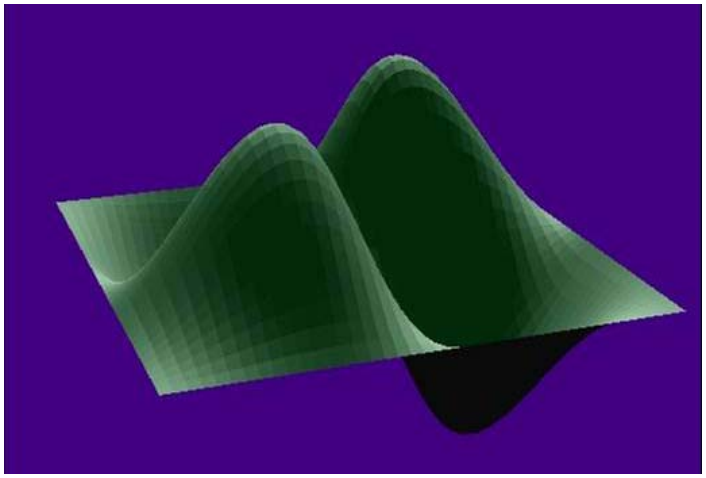


Figure 16. AVEC Surface Class Rendering

7.2 Geometric Entities

New geometric entities will naturally arise with AVEC. Figure 16 is a depiction of an instantiated Surface class as rendered in AVEC. This is a Hermite surface based on a rectangular matrix of coordinates and vectors. A special AVEC_Type was developed to support Surface development. The class Point_on_Surface contains both coordinates and three vectors. As indicated in Figure 16, a Hermite surface formulation is very adaptable.

The number of geometric entities required to render conceptual and preliminary design features is small. For one dimensional constructs, we require line segments and space curves. For two dimensional constructs, we require a planar patch and a Hermite (cubic) parametric surface. Classes for higher order geometric assemblies are in development. These include contour class to interpolate a surface between a series of curves. Classes to address Boolean intersection and

trimming will become valuable. Where computational design is one's business, meshing and geometry are very closely linked. One should not be developed without the other in mind. While AVEC will be developed with a small number of native geometry classes, AVEC is designed to facilitate the integration of any geometry kernel and meshing utilities.

The computational designs reported in Reference [9] required only Hermite curves and surfaces and did NOT require complex geometric operations. The model surfaces were generated with simple interpolation between curves. The tool set used in support of Reference [9] did not provide this interpolation for the joined-wing topology. The class that was developed for Reference [9] has been incorporated into AVEC as Contour class. All indications lead to the conclusion that powerful commercial CAD or expensive geometry kernels are NOT required to develop geometry for the SensorCraft application described here in Section 2 and 3. It is more important to have access to source code using relatively simple geometric constructs described in a number of standard textbooks. This is an important lesson based on significant experience.

7.3 The Geometry Viewer

Viewing orientation of geometry in the AVEC graphical pane is controlled through the main view menu with options for (a) translate (b) rotate (c) zoom (d) adjust viewing volume (e) set camera coordinates (f) set target coordinates. All action is based on standard OpenGL viewing constructs. Translations shift camera and target simultaneously. Rotations take place in local spherical coordinates. Zoom responds with exponential action (slow or fast depending on distance between camera and target).

7.4 Save and Restore Features

AVEC models can be saved and restored in XML format. The data structure is preserved, including all parent/child relationships between components and all master/slave dependencies between variables. Creating the save function is a simple process. Model restoration is much more challenging to program and requires two sweeps. The first sweep establishes parent/child hierarchy. The second sweep establishes master/slave dependencies.

AVEC provides the ability to save derived classes. A derived class is constructed from an instantiated AVEC class in terms of the parent object and all the children progeny and with variable dependencies declared. A derived class is saved in XML format but without enumerating the independent variables. The derived class can be instantiated as a part of any subsequent model. For instance, the Enhanced_Airfoil class will be interactively constructed from the basic Airfoil class with class Curve instantiated as child objects. This multi-level class will be saved in XML format for recall as part of other models that require the Enhanced_Airfoil class. Dependency paths are reassigned relative to the restored point in the model tree.

The functionality used to save and restore models and derived-classes in AVEC will also support a future copy-object() function. However, this will be addressed after some significant level of analysis management has been developed in AVEC. This will drive the need for database management. (Reference 18 - XML)

Currently, AVEC development depends on an awkward process of compilation and testing within function AVEC_Main_Test_Init( ) called by the main calling program. Future versions of AVEC will benefit with model descriptions based on simple textual file input that describe class in terms that are far simpler for a programmer to construct than XML format.

8. Ongoing Developmental Needs

8.1 Integrating Object-Oriented Vehicle Design with Performance Assessment

Each specialist in a design process can identify their function and the data input they need to perform their task. If a design team is brought together in a sharing process to develop a design process for any concept, a waterfall diagram (e.g. Figure 6) will naturally arise. The blocks on their waterfall diagram will indicate the AVEC classes that require development. AVEC classes that are developed for one project should be developed with re-use in mind. A notional design process for SensorCraft was introduced in terms of the waterfall diagram presented in Figure 6.

The capability cited in Reference [9] is the prototype application that guided Figure 6 and subsequently the pilot AVEC environment. In addition to the various geometric models and associated meshes, the computational design model addressed geometrically non-linear structures, high-order linear panel method to model follower forces, trimmed aeroelastic equilibrium, allocated fuel consumption in a coarsely defined mission mode, and structural optimization.

For instance, the mission class of Reference [9] is represented in Figure 6 as a combination of (A) Events Manager and (B) System Power Assessment. The output of block (A) is (1, 2, 3) Altitude and Mach which in turn feeds (B) Power Assessment, (H) Drag and Range Solver and (I) Maneuver Trim. The output of block (B) is (5) which in turn feeds (H) Drag and Range Solver. Block A input includes (4) Require Thrust and (19) Control Surface Settings. Block B input includes (9) Power System Position and (17) Fuel Consumed.

Clearly, a mission class will be developed in AVEC to manage all aspects as described above. The mission class will be designed to interact with the air vehicle class. Actually, the initial mission class will be very simple to construct.

Where we have many design variants to be analyzed in one discipline such as structures, it does not make sense to instantiate a separate solver class for each design. However, it does make sense to instantiate a separate analysis model for each variant. Data from an analysis model will be sent to a common solver object that would manage several cases simultaneously. Thus, block L (Structural Equilibrium) in Figure 6 could represent a structures model that will be placed in a common queue for solving structures equations. The management of the queue and the associated data will be a challenge to program. The first analysis class to be formulated will address aerospace structures.

Equivalent beams and plates require virtually no computational mesh. Reference [15] is an example of equivalent-plate modeling applied to a joined-wing concept. These “equivalence” methods are closely related to the P-version of finite element modeling. A structural designer working at the conceptual level might consider developing a C++ class for equivalent plate, or more generally, equivalent P elements.

A traditional FEM analysis employs a comprehensive mesh that must be regenerated for any geometric change. As an alternative to the comprehensive mesh, one could mesh parts independently. Interface elements are “sewn” together after-the-fact. This approach is reported in literature as interface elements.

Decisions have yet to take place on methods of meshing. Indeed, AVEC, when matured, may be a wonderful open environment for exploring the various issues involved with meshing such as unstructured solvers, convergence and interface elements. Meshing development is closely linked with geometry development. As indicated, serious geometry development will be enhanced when a geometry kernel is integrated with AVEC.

AVEC will ultimately address various optimization algorithms integrated with geometric non-linearity, follower forces and aeroelastic trim. There is room to improve on the fully-stressed optimization presented in Reference [9]. One would avoid separate serial cyclical convergences procedures if the aerodynamic (loads and trim) model and non-linear structural model were solved simultaneously.

8.2 The Management of Large Datasets

Computational design models with many design variants requires many analysis models each with potentially large datasets of various sorts. A computational design environment must provide a practical database class to facilitate the various types and functional requirements. The optimal solution is not intuitively obvious. The question for computational design is how good is good enough. Whatever solution we use today can always be improved in future developments.

The data models in AVEC are dependency tracked and therefore require some in-kind dependency-management that extends into the database. The data models in AVEC are hierarchical. This hierarchical form must be efficiently stored in the database, but does not prescribe the database format itself. AVEC classes are somewhat invariant in their form with a fixed number of variables. This feature will facilitate database development. For very large models with many design variants, the database class will benefit with ability to geographically distribute itself among a number of storage devices.

A large family of design variants was used in Reference [17]. The process would have improved tremendously with a proper database. As it was, the database was a large collection of input and output files. However time-consuming, the response surface models such as Reference [17] were successful in transforming results from large data sets into a single algebraic form (i.e. response surface) that was subsequently optimized.

As an interesting note: Reference [18] contemplates the possibilities of managing large data models under XML. While XML is not appropriate form for storing numerically intense datasets, it might be appropriate as an interface and exchange medium.

8.3 Open-Source Development

The business of open-source development is addressed in References [19] and [20]. In today's software development market, the question is not whether to engage in open-source development – but how much and what part of the development should be open-source. This is true, even for today's established commercial products[6]. Where research is a goal, open-source integration tools have the potential to serve a significant role.

When AVEC appears attractive to a small team of collaborators, the next stage of development would seek to serve a large community of users with a library of C++ (or equivalent) classes that

[6] New York Times (International Herald Tribune) 25 January 2006, "Microsoft to Disclose Parts of Windows Source Code", by James Kanter.

could be instantiated or inherited and modified. Open source software must be extremely reliable, relevant and documented to reach a large user base that is willing to use it and support it. Funding mechanisms should follow the development of a creative business strategy. For instance the AFRL can develop funding plans under SBIR contracts and collaborative commitments under CRDA mechanisms. Many open-source business issues are addressed in Reference [20]. Open source software development follows the following steps

1) Identify a need.
2) Develop Pilot Code
3) Create User Requirements
4) Develop a (small and tight) dedicated team of Software Developers who share the (identified) need (AFRL funding perhaps)
5) Develop Software Requirements
6) Develop production software library of class structure
7) IP Legalities (Licensing Limitations)
8) Publication and Distribution
9) Applications

Successful development requires a tight nucleus of highly trained and inspired developers who can deliver a reliable prototype (pilot) code that can be appreciated by a relatively large base of computational designers. Software innovators can be guided by design engineers who develop innovative designs.

9. AVEC Distribution

The AVEC system is contained and managed by a top level C++ class. Subordinate classes are designed to work with a GUI that is compiled and integrated for inheritance by the Level II Programmer or the Level III End User. Integration with other object libraries (e.g. geometry kernel) to append new functionality to the native Component class. The creation of such a environment that is also distributable required some additional effort beyond the initial isolated development.

The AVEC distribution disk has been shown to work for any of the three user types described in Table 2. These distribution files are described below. However, as indicated in the discussion on open-source, AVEC requires significant testing maturing within an AFRL MultiDisciplinary Technology Center Consortium before sharing with the public.

9.1 End User

The following directories are managed by the AVEC End User.

AVEC_Data: This directory contains all the files necessary to save and restore AVEC models and derivative class structures. The location of these files is user-specified by the end-user in the path_file. AVEC automatically searches out the path_file where data file structures are retained. The end user must tailor the path_file so AVEC can find where all key files are retained. Here is an example:

```
ROOT                NULL              C:\\
USER_HOME           ROOT              USERS\\blairm\\
AVEC                USER_HOME         AVEC\\
AVEC_STEP           AVEC              AVEC_03_Release_Export\\
APPLICATIONS        AVEC_STEP         AVEC_Applications\\
AVEC_DATA           AVEC_STEP         AVEC_Data\\
AVEC_MODELS         AVEC_DATA         AVEC_Model_Data\\
AVEC_AIRFOILS       AVEC_DATA         AVEC_Airfoil_Library\\
AVEC_TRACK_FILE     AVEC_DATA         avec_track_file.wri
AVEC_CLASS_LIB      AVEC_DATA         AVEC_Class_Lib\\
AVEC_GLOBAL         AVEC_DATA         AVEC_Global_Data\\
AVEC_UNIT_TABLE     AVEC_GLOBAL       units_conversion_factors.txt
ASTROS              APPLICATIONS      astros\\
```

The first column represents key variables that are compiled with AVEC. Each row contains a key variables in column 1 that is assigned a string value (file path) that is a combination of the second column (interpreted) and the third column. Key variables in the second column refer up to strings represented by key variables in the first column.

AVEC_Applications: This directory conveniently contains 3rd party application software.

9.2 C++ Programmer

In addition to the above path_file setup, the programmer is provided with the data files required to compile and link new derivatives of the Component class without requiring direct GUI license support.

AVEC_Component_Derivatives: Contains sample classes that are derivatives of class Component. The idea is for the C++ Programmer to create their own customized Component derivatives that are compiled and incorporated into the object library along with all other object libraries resident in lib.

AVEC_Main: The main executable that instantiates the avec class and may be customized in how avec interacts with other external functionality. For instance, test code may be created that drives avec functions. Specifically, the C++ Programmer may want to test new Component derivatives without instantiating the GUI interface.

includes : Contains all class definitions required to compile derivative Component classes.

lib: Contains all object code with classes required to develop new derivatives of class Component and drive AVEC in general.

9.3 AVEC Programmer:

In addition to the C++ Programmer distribution, the AVEC Programmer receives the fill set of object libraries contained in the following hierarchical data entities.

AVEC_Units: Manages classes related to units of measurement

AVEC_Universal: Manages simple C functions that reside outside AVEC C++ structures.

AVEC_Data_Types: In addition to native C++ data types, AVEC manages a few additional data types that allow interactive I/O with ascii script. Each of these data types is a class structure that directs how to form ascii text from variable data.

AVEC_Base: AVEC_ Base was constructed to house base classes that are universal to AVEC but whose complete definition depends on heirarchical classes defined below.

AVEC_IO: AVEC IO is managed through these class structures.

AVEC_Component_Kernel: These are the base classes that are common to all classes that derive from class Component.

AVEC_GUI: Contains all GUI (i.e. QT derivative) classes for interactive access to AVEC model developments.

AVEC_Kernel: Contains all base classes up to and including the main AVEC class.

AVEC_OpenGL: Contains the interface between OpenGL directives called by AVEC and the AVEC GUI

The above libraries are designed to support Computational Design activities with the functionality described in this paper. Although the AVEC system would benefit with the functionality of dynamic linking, the complexity involved was more than could be justified by the author at this point. Indeed, it may not be practical for a compiled programming environment.

10. Conclusions

AVEC is the pilot code for an integration tool that serves a design research team with a library of classes that can be inherited along with standard C++ code. AVEC represents the beginning of an open-source collaboration that addresses common basic needs such as dependency management, graphical rendering etc. Collaborators benefit with the ability to conduct basic design research in “whatever” without getting lost in a myriad aspects of mundane computer technology. The choice of C++ serves the computational community best with highly reliable compiled code. Technology transition for integration software is best realized with open-source distribution. The open-source approach requires some careful attention to protect the integrity of the capability. AVEC represents a possible prototype towards the establishment of open-source integration software that supports Computational Design directly and AFRL technology program indirectly.

11. References

[1] Maxwell Blair, “Computational Design Challenges for Non-Linear Aeroelastic Systems”, International Forum on Aeroelasticity and Structural Dynamics”, 28-30 June 2005, Munich Germany IF-145.

[2] John D. Binder, “Knowledge-Based Engineering – Automating the Process”, Aerospace America, Vol 34, pp 14-16, March 1996

[3] Ilan Kroo, “An Interactive System for Aircraft Design and Optimization”, AIAA-92-1190, AIAA Aerospace Design Conference, 3-6 February 1992, Irvine CA.

[4] Brett Malone, Scott Woyak, “An Object-Oriented Analysis and Optimization Control Environment for the Conceptual Design of Aircraft”, AIAA-95-3862, 1st AIAA Aircraft Engineering, Technology, and Operations Congress, 19-21 September 1995, Los Angeles CA.

[5] Hongman Kim, Brett Malone, Jaroslaw Sobieszczanski-Sobieski, “A Distributed, Parallel, and Collaborative Environment for Design of Complex Systems”, 45th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, 19-22 April 2004, Palm Springs, CA

[6] David Rodriguez, Peter Sturdza, “A Rapid Geometry Engine for Preliminary Aircraft Design”, AIAA-2006-0929, 44th AIAA Aerospace Sciences Meeting and Exhibit, 09-12 January 2006, Reno NV

[7] Jan Vandenbrande, Thomas A. Grandine, Thomas Hogan, “The Search for the Perfect Body: Shape Control for MultiDisciplinary Design Optimization”, AIAA-2006-0928, 44th AIAA Aerospace Sciences Meeting and Exhibit, 09-12 January 2006, Reno NV

[8] The DAKOTA Project: Large-scale Engineering Optimization and Uncertainty Analysis, [http://endo.sandia.gov/DAKOTA/]

[9] Maxwell Blair, Robert A. Canfield, Ronald W. Roberts Jr., "Joined-Wing Aeroelastic Design with Geometric non-Linearity", AIAA Journal of Aircraft, Vol 42, Number 4, July-August 2005, pp 832-848.

[10] Johnson, F. P. "SensorCraft." AFRL Technology Horizons®, vol 2, no 1 (Mar 01): 10-11 URL: http://www.afrlhorizons.com/Briefs/Dec04/VA0308.html

[11] Ryan Craft, “Drag Estimates for the Joined-Wing SensorCraft”, Master of Science Thesis, Air Force Institute of Technology (AFIT/ENY), June 2005.

[12] Ben Smallwood, “Structurally Integrated Antennas on a Joined-Wing Aircraft”, AIAA-2003-1459, 44th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, 7-10 April 2003, Norfolk VA.

[13] Daniel D. Strong, Raymond M. Kolonay, Frank E. Eastep, Pete M. Flick, “Flutter Analysis of Wing Configurations Using Prestressed Frequencies and Mode Shapes”, International Forum on Aeroelasticity and Structural Dynamics”, 28-30 June 2005, Munich Germany.

[14] Richard D. Snyder, JiYoung Hur, Daniel D. Strong, Philip S. Beran, “Aeroelastic Analysis of a High-Altitude Long-Endurance Joined-Wing Aircraft”, International Forum on Aeroelasticity and Structural Dynamics”, 28-30 June 2005, Munich Germany.

[15] Luciano Demasi and Eli Livne, “Exploratory Studies of Joined-Wing Aeroelasticity”, AIAA-2005-2172, 46th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, 18-21 April 2005, Austin TX.

[16] Keith Hunten, Collin McCulley, Anontio De La Garza, Maxwell Blair, “The Application of the MISTC Framework to Structural Design Optimization”, AIAA-2005-2127, 46th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, 18-21 April 2005, Austin, TX.

[17] Cody C. Rasmussen, Robert A. Canfield, Maxwell Blair, “Joined-Wing Sensor-Craft Configuration Design”, AIAA-2004-1760, 45th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, 19-22 April 2004, Palm Springs, CA

[18] Risheng Lin, Abdollah A. Afjeh, “An XML-Based Integrated Database Model for MultiDisciplinary Aircraft Design”, AIAA Journal of Aerospace Computing, Information, and Communication, Vol 1, March 2004

[19] Open Source Initiative: http://www.opensource.org/

[20] Carolyn A. Kenwood, “ A Business Case Study of Open Source Software”, The MITRE Corporation under Army Contract DAAB07-01-C-C201 http://www.mitre.org/work/tech_papers/tech_papers_01/kenwood_software/kenwood_software.pdf

NOMENCLATURE

AFRL	Air Force Research Laboratory
AVEC	Air Vehicle Environment in C++
DFW	Dependent Function Wrapper
DVW	Dependent Variable Wrapper
GUI	Graphical User Interface
HALE	High Altitude, Long Endurance
LMB	Left Mouse Button
MDT	MultiDisciplinary Technology
OOP	Object-Oriented Programming
RMB	Right Mouse Button
SBAAT	Scenario-Based Affordability Assessment Tool